Università di Catania

DISTRIBUTED SYSTEMS AND BIG DATA
A.A. 2024/2025

HOMEWORK 2

Studenti:
Fabio Gallone 1000001752
Claudio D'Errico 1000004015

Abstract (Hw2)

L'applicazione, sviluppata a microservizi, si propone come obiettivo quello di essere resiliente, scalabile e manutentibile per la gestione degli utenti e l'analisi di dati finanziari relativi ai ticker in tempo reale. Il flusso principale parte da un **Client gRPC**, in cui l'utente finale può interagire inviando richieste e ricevendo risposte immediate dal server gRPC.

Il **Server gRPC** è infatti il cuore del sistema per quanto riguarda le operazioni di business. Implementa il pattern CQRS (Command Query Responsibility Segregation), separando le operazioni di scrittura (ad es. registrazione, aggiornamento ticker, cancellazione utente) da quelle di lettura (recupero dell'ultimo valore finanziario, calcolo della media degli ultimi N valori). Questa separazione logica semplifica la manutenzione e consente eventuali ottimizzazioni future. Inoltre, il Server gRPC adotta una politica **At-Most-Once** tramite l'utilizzo di request_id univoci e memorizzati in cache, garantendo che, se il client ritenta una richiesta non andata a buon fine, la stessa operazione non venga eseguita due volte, evitando così duplicazioni e sprechi di risorse.

Il **Database PostgreSQL** memorizza i dati relativi agli utenti (inclusi i ticker e le soglie high/low opzionali) e i dati finanziari (valori storici per ciascun ticker). Per mantenere il database efficiente, un Data Cleaner interviene periodicamente, eliminando i record storici più vecchi e mantenendo solo un numero massimo di valori recenti (ad esempio 20) per ciascun ticker. Ciò garantisce di non sovraccaricare il server, permettendo quindi un'ottima performance delle query, anche con un numero crescente di dati.

La parte di aggiornamento dei dati finanziari è gestita dal **Data Collector**, un servizio che interroga continuamente l'API esterna yfinance. Per rendere il sistema resiliente ai guasti della fonte esterna, è stato implementato un **Circuit Breaker:** in caso di ripetuti errori nel contattare yfinance, le richieste vengono temporaneamente sospese, evitando inutili tentativi e prevenendo il sovraccarico del sistema. Quando il Data Collector riesce ad aggiornare i dati nel database, non comunica direttamente con il Server gRPC o con altri componenti, ma produce un messaggio su un topic Kafka, to-alert-system. Questo passo introduce l'asincronia e disaccoppia la raccolta dei dati dal controllo delle soglie, rendendo l'architettura più flessibile.

Il **Kafka Broker** entra in gioco come sistema di messaggistica asincrona che permette la comunicazione tra i servizi interni senza vincolarli a interagire direttamente tra loro. Nel nostro caso, il Data Collector produce un messaggio sul topic to-alert-system quando i dati sono aggiornati, e **l'AlertSystem** consuma questo messaggio per sapere che ci sono nuovi dati da analizzare. L'AlertSystem, ricevuta la notifica di aggiornamento, interroga il Database per verificare se le soglie high o low definite dagli utenti sono state superate. Se questa condizione è verificata, l'AlertSystem produce un nuovo messaggio sul topic to-notifier.

A questo punto interviene **l'AlertNotifierSystem,** che consuma i messaggi dal topic to-notifier. Ogni messaggio contiene le informazioni necessarie per individuare l'utente e il ticker interessato. L'AlertNotifierSystem, senza interagire con altri componenti, invia direttamente un'email all'utente per avvisarlo che la soglia è stata superata.

Completa il quadro il **Zookeeper**, un componente di coordinamento utilizzato dal Kafka Broker per gestire lo stato del cluster, le partizioni, le repliche e l'elezione dei leader. Zookeeper non interagisce direttamente con i servizi applicativi come il Data Collector o AlertSystem, ma lavora dietro le quinte per mantenere la stabilità e la coerenza di Kafka.

Rispetto l'homework1, sono stati aggiunti i seguenti componenti:

- **Il server** adotta adesso il pattern **CQRS** (Command Query Responsibility Segregation) separando logicamente le operazioni di scrittura (ad es. RegisterUser, UpdateUser, DeleteUser) da quelle di lettura (GetLatestValue, GetAverageValue), rendendo l'architettura più chiara.
- **Kafka Broker:** funge da sistema di messaging asincrono. Riceve i messaggi di aggiornamento dal Data Collector (topic to-alert-system) e i messaggi di alert dall'AlertSystem (topic to-notifier).
- **AlertSystem:** consumando i messaggi to-alert-system da Kafka, controlla nel database se i valori sono maggiori o inferiori delle soglie high/low definite per gli utenti. Nel caso in cui si verifichi una delle due condizioni produce un alert su Kafka (topic to-notifier).
- **AlertNotifierSystem:** consumando i messaggi to-notifier da Kafka, invia email agli utenti interessati per informarli che il valore del ticker risulta essere maggiore o inferiore al valore delle soglie (High/Low) indicate.

Diagramma architetturali

- **Server gRPC:** rimane invariato
- **Data Collector:** raccoglie dati (con Circuit Breaker verso yfinance) e aggiorna il Database, poi produce un messaggio su Kafka (to-alert-system).
- **Kafka Broker:** nodo di messaggistica che ospita i topic "to-alert-system" (da Data Collector verso AlertSystem) e "to-notifier" (da AlertSystem verso AlertNotifierSystem)
- **AlertSystem:** consuma da "to-alert-system", controlla le soglie nel Database e se superate produce un messaggio su "to-notifier".
- **AlertNotifierSystem:** consuma da "to-notifier" e invia email all'utente

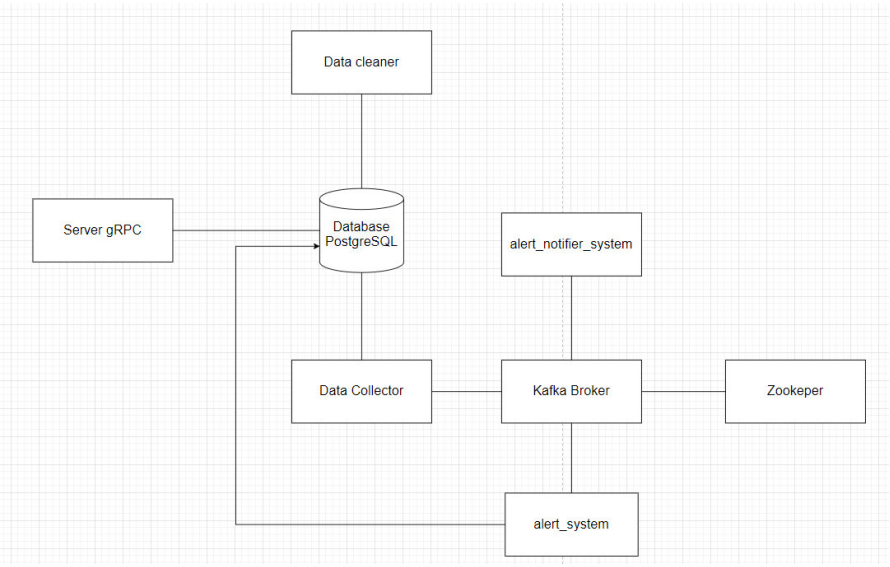
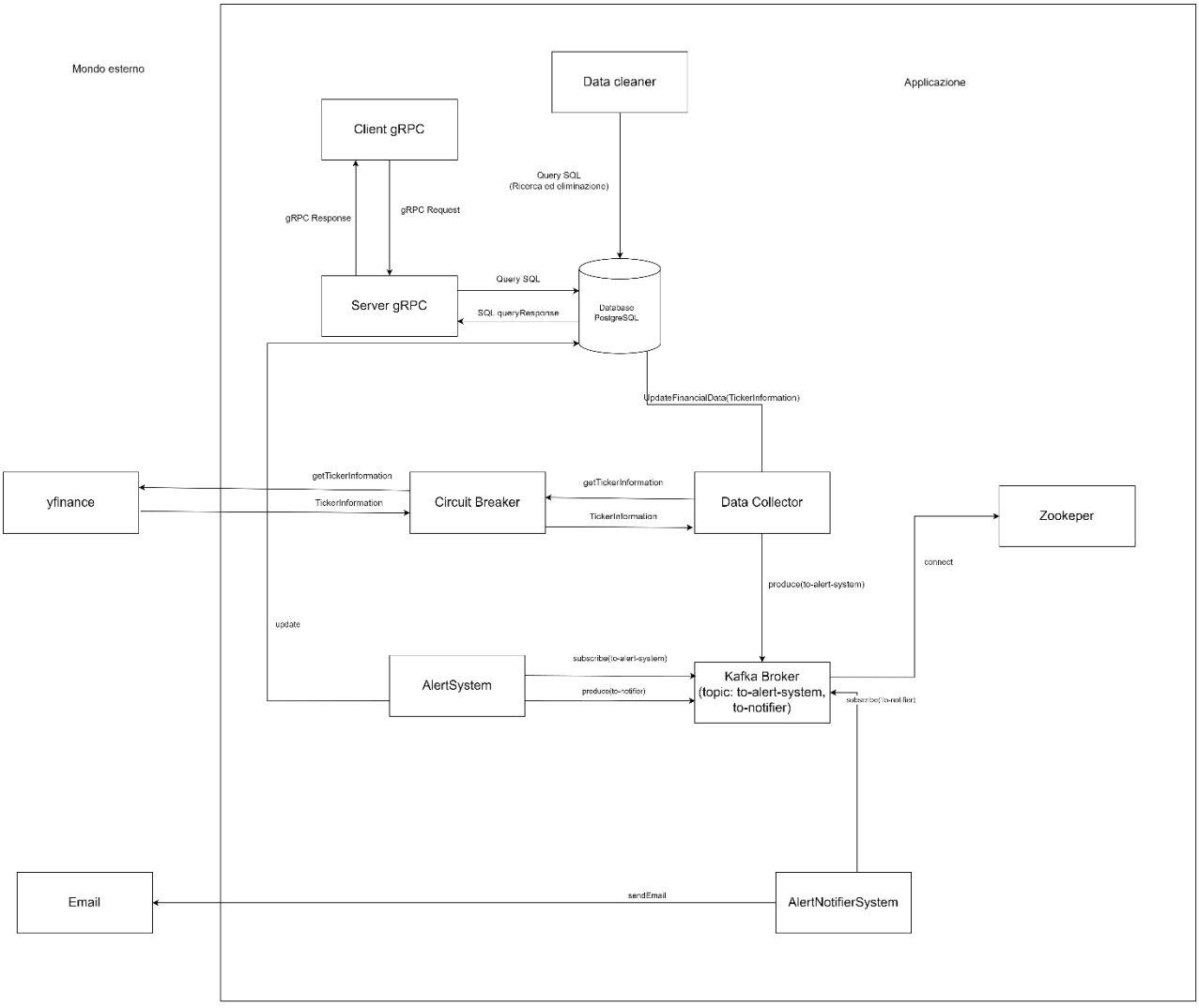


Diagramma delle Interazioni

Le novità principali di questo homework per le interazioni tra i componenti sono:

- **Data Collector:** comunica con Kafka tramite il topic “to-alert-system”.
- **Alert system:** questo componente comunica con Kafka (consuma da “to-alert-system”). L’Alert system rileva il segnale di aggiornamento inviato dal Data Collector e ricevuta la notifica interroga il database per verificare se i valori appena aggiornati superano le soglie definite dagli utenti. Se le condizioni di alert sono soddisfatte, produce un messaggio di allerta sul topic “to-notifier” di Kafka.
- **Alert notifier system:** legge dal topic “to-notifier”, riceve un messagio di alert e si preoccupa di inviare una mail all’utente per avvisarlo se il valore del ticker risulta essere maggiore o inferiore al valore delle soglie (High/Low) indicate.
- **Zookeeper:** non interagisce con i sistemi applicativi (Data collector, Alert system, Alert notifier system) ma fornisce le informazioni di coordinamento al broker Kafka che si connette a Zookeeper per ottenere e aggiornare le informazioni sullo stato del cluster, assicurando la corretta distribuzione e replica dei dati.



Lista delle API implementate

Nome API	Descrizione	Messaggio di Richiesta	Messaggio di Risposta
RegisterUser	Registra un nuovo utente con un ticker di interesse.	**RegisterUserRequest** – email (string) – ticker (string) – request_id (string) – high_value (double) – low_value (double)	**RegisterUserResponse** – message (string)
UpdateUser	Aggiorna il ticker di un utente esistente e/o le soglie high/low. Se il ticker cambia e non vengono fornite nuove soglie, queste vengono resettate a NULL.	**UpdateUserRequest** – email (string) – ticker (string) – request_id (string) – high_value (double) – low_value (double)	**UpdateUserResponse** – message (string)
DeleteUser	Elimina l'account di un utente.	**DeleteUserRequest** – email (string) – request_id (string)	**DeleteUserResponse** – message (string)
LoginUser	Esegue il login di un utente tramite l'email.	**LoginUserRequest** – email (string)	**LoginUserResponse** – message (string) – success (bool)
GetLatestValue	Recupera l'ultimo valore disponibile per il ticker dell'utente.	**GetLatestValueRequest** – email (string)	**GetLatestValueResponse** – email (string) – ticker (string) – value (double) – timestamp (string)
GetAverageValue	Calcola la media degli ultimi X valori per il ticker dell'utente.	**GetAverageValueRequest** – email (string) – count (int32)	**GetAverageValueResponse** – email (string) – ticker (string) – average_value (double)